No Loitering: Exploiting Lingering Vulnerabilities in Default COM Objects

David Dewey†‡
dewey@us.ibm.com

Patrick Traynor‡
traynor@cc.gatech.edu

†Advanced Technology Group
IBM Security

‡Converging Infrastructure Security (CISEC) Lab
Georgia Institute of Technology

Abstract

The Component Object Model (COM) facilitates the creation of software plugins for applications running in Microsoft Windows. ActiveX is a common instantiation of this infrastructure, and uses COM to create plugins for Internet Explorer. As vulnerabilities in COM objects included in the installation of Windows have been found, Microsoft has responded by blacklisting their use by specific applications. In this paper, we demonstrate that the defense mechanisms protecting vulnerable COM objects can be easily circumvented. Specifically, our attack exploits systemic transitive trust among COM objects and allows for the instantiation and exploitation of any of several hundred known flawed controls. After demonstrating this weakness on fully patched Windows XP, Windows Vista and Windows 7 machines, we design and implement a system-wide enforcement architecture called COMBlocker, which checks the instantiation of COM objects against a global policy. We then show that COMBlocker is an effective mitigation for such attacks while imposing minimal overhead (approximately 0.5ms per policy check). In so doing, our techniques make the exploitation of default COM objects significantly more difficult.

1 Introduction

The Component Object Model (COM) is a language-neutral design philosophy allowing for the creation of discrete software components that can be integrated into other applications. Every COM object exposes a standard interface, allowing developers to create extensible applications that provide a well-defined plugin architecture. Others developers can leverage these standard interfaces to create COM objects that can themselves be consumed by additional applications, allowing for more creative and enhanced solutions to user demands. ActiveX extends this architecture to browsers and allows for such extensible design to be expanded to web-based applications. Through ActiveX, a web developer can force the execution of native code in the context of the browser through the COM-based plugin infrastructure in Internet Explorer.

Such extensible functionality has frequently been targeted for malicious purposes [19, 20, 22–24]. To aid in securing ActiveX, a configuration policy known as the "killbit" list was added to Internet Explorer. This allows users and vendors to manage a list of controls that should never be loaded by this particular browser. In the event that a vulnerability is discovered in a control, the killbit list can be used to prevent its instantiation in IE. By adding vulnerable controls to this list, a popular attack vector used by hackers can be mitigated. The killbit list also serves as the basis for other security policies in a small number of additional applications. For instance, Microsoft Office applications will prompt the user if a document has a COM object embedded in it that is listed in the killbit list.

In this paper, we demonstrate that this mitigation mechanism is insufficient to protect systems against the exploitation of vulnerable COM objects. Specifically, implementing and enforcing security policies through the killbit list on a per-application basis is an inadequate means of mitigating attacks against vulnerable COM objects that are part of the default operating system installation. We show that individual applications do not have a broad enough view into the behavior of the COM objects they load to effectively enforce a security policy.

In this paper, we make the following contributions:

- **Discover and characterize a systemic weakness in the COM security infrastructure:** We demonstrate that the existing killbit list security policies governing the instantiation of COM objects can easily be circumvented. We show that due to a weakness in the underlying COM architecture, many COM objects that are part of the default in-

stallation of the operating system can load other objects without ever consulting a policy of any kind. We demonstrate that this attack is possible through *virtually every* application that is commonly installed on Windows.

- **Confirm the ability to exploit several hundred known flawed and vulnerable COM objects installed by default in Windows:** Over the past several years, the common response to security vulnerabilities reported in COM objects has been to use existing policy mechanisms to prevent their instantiation in a few discrete applications. Because the COM objects themselves are often not corrected, our circumvention identifies a significant security risk. It is important to note that the attack described in this paper takes advantage of controls that are already installed on Windows and does not require the victim to install or load a control created by the adversary.

- **Design and implement a prototype policy enforcement infrastructure for COM objects:** We design and implement a prototype infrastructure for the system-wide COM instantiation policy enforcement, which we call COMBlocker. In this system, we are able to quickly compare (average lookup time of 554μs) the instantiation of COM objects against a global policy. Our approach is effective as it uses binary rewriting to force all COM object instantiations to be compared against the global policy, thus mitigating the above attack. We then compare our proposed solution to the patch recently issued by Microsoft (Security Bulletin MS10-036), which was created in response to our private disclosure of the vulnerability.

Through the security weakness documented in this paper, *we demonstrate that Windows is susceptible to attack against several hundred known flawed controls, which are already resident on the system.* Many of the vulnerabilities that are left "loitering" on Windows due to unpatched COM objects, when exploited, allow adversaries to execute arbitrary code. These vulnerabilities can potentially be exploited to install malware. In demonstrating the weaknesses of the existing security mechanisms associated with COM, we show that adversaries have a large number of existing vulnerabilities to target that can provide them with complete control of a system.

The remainder of this paper is organized as follows: Section 2 discusses important related research efforts; Section 3 provides background information related to COM objects and their extensibility; Section 4 illustrates the process by which the killbit list can be bypassed, thereby reenabling the ability to exploit hundreds of previously documented vulnerabilities; Section 5 describes our mitigation architecture, which creates a centralized point for COM object policy enforcement; Section 6 offers experimental results and discussion; Section 7 provides concluding remarks.

2 Related Work

ActiveX has garnered significant attention from security experts over the past several years. Attacks against these controls range in severity from downloading files to an adversary-supplied location to arbitrary code execution [19, 20, 22–24]. As a variety of vulnerabilities have been found in COM objects included with the default installation of their operating systems, Microsoft has generally prevented the exploitation of the objects by adding them to the killbit list. Dowd, Smith, and Dewey demonstrated that it was possible to bypass the killbit settings in Internet Explorer using a vulnerability for ActiveX controls [4]. However, while this weakness was patched for Internet Explorer [25], the general susceptibility of the COM architecture and applications relying upon it has not previously been investigated.

The issue of securing the execution of content published by potentially untrusted third parties is not unique to COM or Windows. Java applets, for example, expose a web browser to similar classes of threats encountered by ActiveX [1, 16]. Security of Java applets has been studied extensively with several solutions proposed to verify the publisher of the content and enforce access rights on the content as it executes. Jaeger, et al. [14] and Islam, et al. [12] developed systems based on public key cryptography to verify the publisher of dynamically downloadable executable content. They both then go on to propose solutions for the enforcement of access controls on the code as it executes.

Security policies have been developed to address whether COM objects should be loaded by Internet Explorer. The killbit list attempts to prevent the instantiation of known bad controls [21]. As documented by Loscocco et al., ActiveX controls can be signed similar to the way that was proposed for Java [15]. Additionally, ActiveX supports the concept of "Safe for Scripting" [17]. This allows a control to tell Internet Explorer whether it can be safely loaded by the script engine. Microsoft has implemented additional security policies governing the instantiation of COM objects in other applications including the MS Office suite [26]. However, policies governing the instantiation of COM objects are implemented by the COM container itself.

Security retrofits to well-established software infrastructures such as COM generally require significant effort. A number of researchers have investigated the

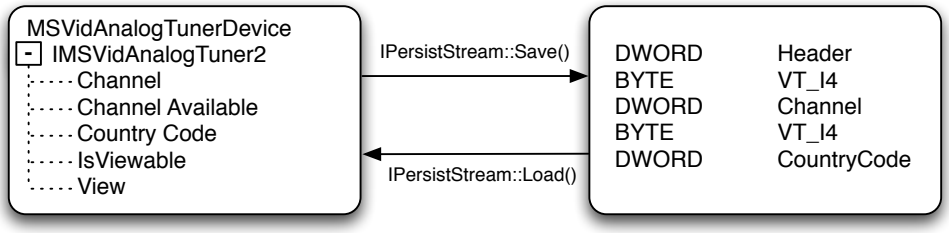

Figure 1. COM objects load and save their state information using the `IPersistStream::Load()` and `IPersistStream::Save()` API calls, as illustrated here by the MSVidAnalogTunerDevice object.

problem of retrofitting legacy systems with security infrastructure. These efforts use a range of approaches including static analysis of code [3, 7, 27, 29, 30] and the monitoring of program behavior to determine security sensitive operations [10, 13]. The work by Ganapathy, et al. demonstrates the injection of authorization policy enforcement code into existing legacy applications including web and proxy servers [9] and the Linux Security Module infrastructure [8]. Fraser, et al. perform a similar deployment of authorization hooks in MINIX [7]. This builds on the concept of inlined reference monitoring as described by Bauer, et al. [2], Erlingsson [5], and Evans and Twyman [6]. Each of these demonstrates how an existing application can be modified to perform functions not originally designed into the code. The closed nature of the COM architecture requires that our solution rely on binary rewriting techniques rather than those used on source code.

3 COM Background

To understand the attack outlined in this paper and our proposed solution, it is necessary to understand the architecture of COM. This section provides a brief overview of COM and how it is implemented.

3.1 Introduction to COM

The Component Object Model (COM) is a language-neutral design philosophy for the creation of software components in the Microsoft Windows operating system [28]. Each COM object must extend the `IUnknown` interface as it is documented by Microsoft. This base interface allows a developer to query an object to learn more about its behavior and how it can be implemented. There are several other interfaces available that can be extended by objects depending on their intended use. One such interface is `IDispatch`. This provides generic methods allowing for the interaction with object-specific methods and properties. This provides functionality similar to the interaction one would expect in other object-oriented architectures.

Each COM object that is installed on the system is listed in the windows registry under a unique identifier called a Class ID (CLSID). On an average Windows system, there are tens of thousands of COM objects registered. An application choosing to instantiate one of these objects simply needs to supply the corresponding CLSID and the desired type of interface to one of the appropriate Win32 APIs. Each of these APIs simply looks up the CLSID in the registry, loads the library listed in the registry for that object, and returns a handle to the object. Whichever interface was specified will allow access to a set of methods and properties corresponding to that interface.

An interesting feature of COM is its ability to persist object state across instantiations. There are a number of interfaces which, if implemented by an object, can be used to save the runtime state and/or resurrect saved state during instantiation. Some examples of these interfaces are `IPersistStorage`, `IPersistStream`, and `IPersistStreamInit`. In each of these cases, the state of the object is serialized and written to disk. Conversely, the serialized data can be read from disk at instantiation time and used to resurrect the object state. It is up to the developer of the object to determine which object properties are to be persisted and their data type. As shown in Figure 1, an object called MSVidAnalogTunerDevice contains the properties Channel and CountryCode. The values of these properties can be set at instantiation using the `IPersistStream::Load()` method or saved to a binary stream using `IPersistStream::Save()`.

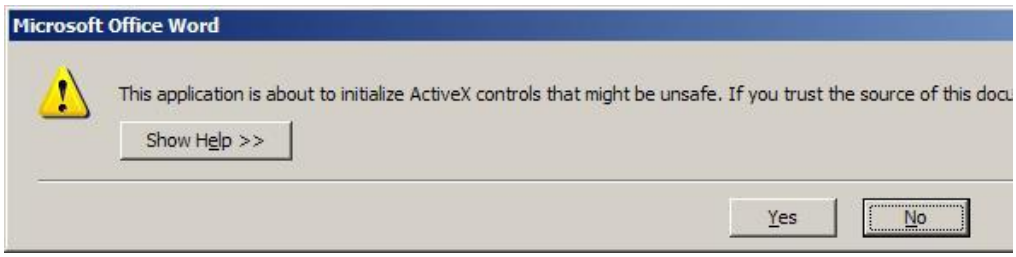

Figure 2. Pop-up Warning from Microsoft Word When Attempting to Instantiate an Out-of-Policy Control

3.2 COM Security

COM exposes some limited information that an application can use in making the determination about an object's security. Applications can instantiate the `IObjectSafety` interface of a COM object if one is supplied by the object. This interface allows the application to determine whether the object defines itself as being "Safe for Scripting" and/or "Safe for Initialization" [17]. Safe for Scripting indicates whether a COM object deems itself safe to be interoperated with through the Internet Explorer scripting engines. Safe for Initialization indicates whether a COM object deems itself safe to be initialized from persisted object state data using the methods described above. In addition to the COM interface, an object can register the same information in the Windows registry. Specifically, two subkeys can be created under the Implemented Categories key under the CLSID for the object The main problem with this interface is that the object itself informs the application about its own security properties. In the absence of any other security-related information, an application has no choice but to believe what the object tells it.

There are several applications on an average Windows installation that allow a third party to determine which COM objects should be instantiated. Internet Explorer, for example, can be told to load a specific COM object that is required for the operation of the page it is displaying. In the context of IE, this is called ActiveX. Another example is Microsoft Office, which allows document authors to embed rich content such as images, movies, or even other documents in the body of a Word document or Excel spreadsheet. In each of these cases, a potentially untrusted third party makes a determination about which COM objects should be loaded by applications on an end user's workstation.

The `IObjectSafety` interface and its registry-based equivalent are not sufficient to provide any reasonable level of protection. Due to the widespread use of Internet Explorer and the ease with which a user can be redirected to malicious content, IE has received the majority of the focus with regard to security from Microsoft. Specifically, a security configuration option known as the "killbit" list was created. The killbit list is a blacklist maintained in the Windows registry that contains the CLSID of every COM object that should never be loaded and interacted with from the IE scripting engines. Microsoft Office will prompt the user if the content being loaded attempts to instantiate a control that is listed in the IE killbit list. The user can accept the prompt and allow the Office application to load the untrusted content, but the default is to disable the control. Figure 2 shows an attempt to load a killbit-listed COM object and the corresponding interception of that request by the operating system.

Blacklisting can be effective at stopping an application from loading a known bad object, but has no impact on the instantiation of objects with an unknown security level. Additionally, it is only effective for applications that choose to implement the policy; it is not enforced across the system as a whole.

3.3 Complexity of COM Management

One may think that as vulnerabilities are disclosed in COM objects, the simple answer would be to remove the object from the Windows registry, or even delete the library containing the object. This approach is not always possible as it relates to COM for several reasons. First, many libraries containing COM objects store more than one object. While there may be one undesirable object in the library, it must remain on the system to support the other objects it contains. Second, many vulnerable COM objects are critical to the operation of required applications and the operating system itself. It is very common for a COM object to misbehave in applications including Internet Explorer or Word, but perform entirely as expected in other applications. Since there is no system-wide mechanism to determine which applica-

tions can load which COM objects, the object must remain registered, and theoretically available to all. Third, COM is massive. As was previously mentioned, there can be tens of thousands of COM objects installed on an average Windows machine. Not surprisingly, the current killbit list is approximately 600 CLSIDs. It is therefore extremely difficult for any user, regardless of skill level, to maintain these lists properly for each application that they run.

4 Vulnerability Characterization

With an understanding of the COM architecture, we now describe the extent to which a class of problems actually extend throughout the all versions of Windows operating system.

4.1 Architectural Weakness

Once a COM object is installed on a system, it is available for instantiation by any application. There is no central policy governing which applications can load which COM objects. As vulnerabilities are disclosed in individual COM objects, there is nothing preventing an unknowing application from loading that potentially harmful control. Given that vulnerabilities in COM objects are most often left unpatched, any application that loads COM objects is a potential target an adversary. Specifically, an adversary can force an application to load one of the previously installed and known flawed controls allowing for exploitation. This allows adversaries to take advantage of controls that already exist on a system rather than attempting to trick their victim into installing an intentionally malicious object. Because many of the vulnerabilities that have gone unpatched allow for the execution of arbitrary code, an adversary can exploit these loitering flaws as an initial infection vector to install malware. Once they have installed their malware, they can gain complete control over the system. Given this exposure, many applications have implemented their own discrete policies dictating which controls they deem to be safe or unsafe. The Internet Explorer killbit list is an example of one such policy.

These policies are only useful for determining which COM objects will be directly loaded by an application. *They cannot guarantee or enforce the behavior of an object once it is instantiated.* An application must therefore trust the behavior of a COM object once it is loaded. If that COM object then loads other COM objects, this trust is transitively extended. As shown in Figure 3, this trust transitivity exposes a critical security weakness with regard to COM. If an application loads a COM object that is within its security policy, it trusts that object to only load other objects that are also in the security

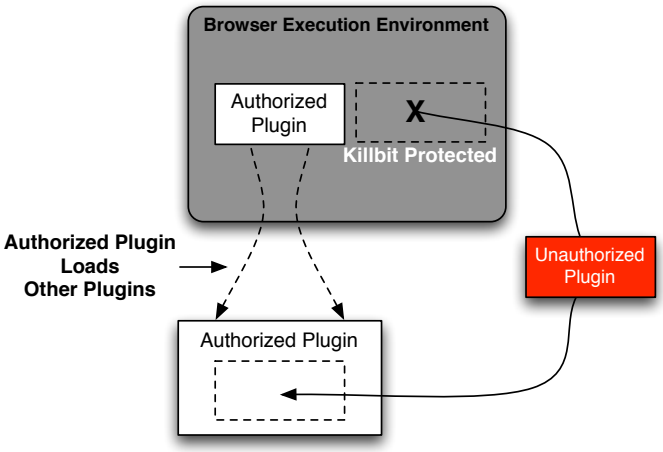

Figure 3. Transitive trust between COM Objects allows normally unauthorized objects to be loaded and executed.

policy. There is no programmatic way for the application to ensure that an object it loads extends its security policy to other objects. It is therefore theoretically possible to exploit this transitive trust relationship in any application that loads COM objects.

To complete an attack exploiting this transitive trust, an adversary requires the following:

1. An application that will render adversary-controlled content.
2. An application that will load COM objects.
3. A COM object that will in turn load other COM objects.
4. A vulnerable object that can be exploited.

Each of these requirements is easily achievable under normal circumstances with an average Windows installation. The following subsections address them in order.

4.1.1 Supplying Adversary-Controlled Content

The first requirement for accomplishing this sort of attack is that an adversary must persuade a target to render content under their control. This is easily accomplished through applications like Internet Explorer and the Microsoft Office suite. End users can be tricked into viewing malicious web pages through a number of means including phishing, cross-site scripting, and content injection. Additionally, users can be emailed Office and other documents or the browser can be used to force the rendering of Office content through the use of ActiveX.

4.1.2 Loading Adversary-Controlled COM Objects

COM objects can be loaded based on content in Internet Explorer, the Microsoft Office suite, as well as any other applications and services commonly installed on Windows that make use of this infrastructure including Flash and Adobe Reader. To force an application such as Internet Explorer to load a COM object, an adversary can supply the CLSID parameter of an <OBJECT> tag. In the case of the Microsoft Office suite, COM objects can be directly embedded in documents using the GUI.

4.1.3 Transitive Trust Amongst COM Objects

The third requirement for the attack is to load a COM object that will in turn load other objects. While this may seem rare, it is actually quite prevalent due to a feature of Microsoft Visual Studio. Because of the complexity of developing COM objects, Microsoft has included with Visual Studio a set of C++ templates called the Active Template Library (ATL). The ATL provides methods for the saving and loading of persistence streams (covered in Section 3.1) so a developer does not have to understand the low-level details. To interact with these methods the developer simply provides a property map, a template defining the order and type of data that is stored in the persistence stream, for the object fields they intend to save stream. This way, as the control reads the persistence stream it understands to which properties it should apply the data.

One weakness of property maps is the ability to define loose types. These are type specifiers that do not directly map to one of the defined types. In other words, the property map can define that data exists in the persistence stream, but the COM object should examine the data itself to determine its type. This is the functional equivalent of a void pointer in the C programming language.

By using the ATL, a COM developer can now easily create a COM object which implements persistence without understanding the underlying required methods or the COM-specific data types with which they are dealing. The critical issue then becomes the handling of loose-typed variants by the ATL-provided methods for loading a persistence stream. To read a persistence stream, the `IPersistStream::Load()` method provided in the ATL will call the method `CComVariant::ReadFromStream()`. As can be seen in the source code for `CComVariant::ReadFromStream()` in Appendix A, there exist two variant types, that if encountered in the stream, will be passed to `OleLoadFromStream()`. An approximation of the source of `OleLoadFromStream()` obtained through reverse engineering the binaries is provided in Appendix B. Here we can see that a CLSID is read from the data stream and passed to `CoCreateInstance()`. The remainder of the data stream is then passed to the `IPersistStream ::Load()` method exposed by the object that has just been loaded.

It is important to note that at no time was a security policy consulted before calling `OleLoadFromStream()`. This code distributed with the ATL can clearly be used to bypass the security policy of the parent application.

4.1.4 Finding a Known Flawed Control

The final requirement for this attack is to be able to load a vulnerable control. Given that vulnerabilities in COM objects are generally left unpatched, several hundred known flawed controls are resident on the average Windows installation. For example, on the system used to write this paper, the killbit list for Internet Explorer is over 600 entries in size. Each of those entries corresponds to known flawed control that is likely still unpatched.

4.2 Proof of Concept Attack

To demonstrate the severity of this attack, and the ease with which it can be accomplished on an average Windows platform, a working example was created leveraging well-known applications and commonly installed COM objects. The attack created for this proof of concept loads a trusted COM object in Microsoft Word, which then loads a known flawed control that is still resident on most installations of Windows. The specific vulnerability used in this example is a known flawed control in all Windows XP installations and results in the execution of arbitrary code when triggered. We note that we tested different vulnerable controls on systems running Windows Vista and Windows 7 and were similarly able to compromise those systems. In this example, the COM object executes shellcode causing Word to listen on a TCP socket. Upon connection to this socket, the adversary is presented with a command prompt.

While this section only describes an attack against Microsoft Word, the same attack was proven to be successful against several other COM containers including Microsoft WordPad, Microsoft Excel, Microsoft Powerpoint and Adobe Reader.

To explain how the proof of concept attack was created, the following describes how each of the requirements in Section 4.1 was met.

1. Microsoft Word was chosen as the parent application for our proof of concept attack. Microsoft Word documents can be easily emailed to users.

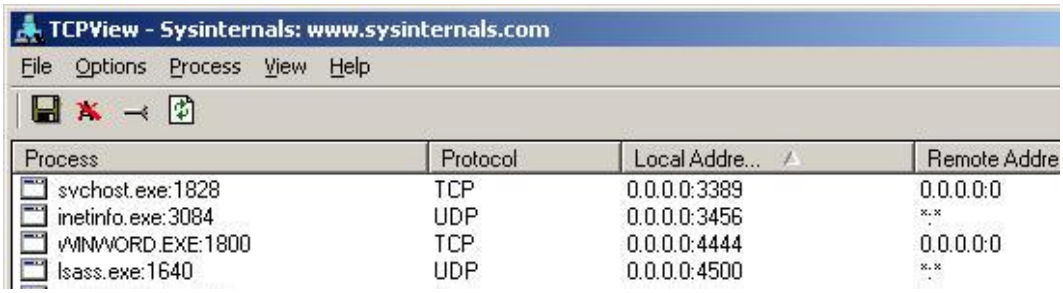

Figure 4. The successful exploitation of a COM Object in Microsoft Word, demonstrated by having Word open a socket on port 4444.

Additionally, the browser can be used as an intermediary in this attack by providing a link to a .doc file or using ActiveX to force Word to open a document of the adversary's choosing.

2. Microsoft Word was chosen for the ease with which COM objects can be embedded in documents. Object insertion is something that is done regularly in the course of authoring a document. This typically comes in the form of inserting images, tables, etc. In many cases these operations are actually embedding COM objects. This same process can be used to insert objects of many different types as long as they conform to the Microsoft Word security policy.

3. The Microsoft Date and Time Picker control was chosen because it provides the functionality to load subsequent COM objects by providing a CLSID in a persistence stream.

4. The final requirement for constructing the attack is to have the COM object loaded in the previous step then load a known vulnerable control. We selected Microsoft's Helper Object for Java, which contains a long standing, unpatched, exploitable vulnerability reachable by instantiating the control outside the Microsoft Java Virtual Machine.

To summarize, we create a working exploit by crafting a Microsoft Word document containing the embedded benign Microsoft Date and Time Picker control. That COM object in turn loads a known flawed control, thus circumventing Word's COM security policy. Once the flawed control is loaded, it triggers the vulnerability, resulting in the execution of arbitrary code. This attack, if executed in the real world, would easily enable an adversary to take full control of a victim's workstation to install malware or use the system for other arbitrarily malicious purposes.

The known vulnerable COM object loaded in this attack is listed in the Internet Explorer killbit list. As such, it is the policy of Microsoft Word to not load this control. Had the attack simply tried to embed the control directly in the Word document, the security policy would have been effective, and the user would have been presented with the dialog shown in Figure 2. While the effectiveness of such warning messages is debatable, our attack allows for the vulnerable control to be loaded without providing the user with any indication that something is amiss. As shown in Figure 4, by using a trusted COM object to load the known flawed control, the security policy is bypassed, and the winword.exe process is now listening on TCP port 4444. This is demonstrable evidence that Microsoft Word has been compromised. In our specific exploit, upon connecting to the socket, the adversary is presented with a command prompt. This allows for the execution of any command in the security context of the user viewing the Word document. In a real attack scenario, an adversary would generally then go on to install their malware – taking complete control of the system.

4.3 Breadth of Attack

The attack described above is not unique to Microsoft Word, the Microsoft Date and Time Picker, and the Helper Object for Java. On the average Windows-based system, there are a large number of applications that would meet the first and second criteria for exploitation. In our testing, identified dozens of COM objects that exist on an average system that were either compiled with the ATL or expose functionally equivalent capability. As previously mentioned, the Windows XP-based system used to write this paper has several hundred known flawed controls still installed. This system is not unique.

Given these numbers, the permutations of application, trusted COM object, and untrusted COM object quickly become unmanageably large. It is therefore unrealistic to require each application to attempt to main-

tain security policies that can reasonably deal with this threat. It is clear that Windows requires an operating system-level security policy governing the instantiation of COM objects.

5 Mitigation Architecture

The lack of a central security policy governing the instantiation of COM objects has been identified as a major source of vulnerability in this paper. In this section, a prototype of a mechanism which we call COMBlocker is proposed that introduces an operating system-level security policy with reference monitor-like functionality. This system will be used to enforce a security policy on the instantiation of all COM objects.

5.1 Design Goals and Assumptions

The goal of COMBlocker is to provide a system-level policy for the instantiation of all COM objects. If every instantiation is monitored by a central policy, the issue of transitive trust can be remedied. The design of COMBlocker assumes that it is attempting to prevent the initial infection vector described in the previous section. It is not designed to secure a previously infected machine. Additionally, the prevention mechanism is designed to secure the instantiation of COM objects by applications that conform to Microsoft's design model. This is not intended to prevent intentionally malicious applications from loading flawed controls. These assumptions follow a common theme: the goal is to prevent the initial attack.

5.2 High Level Architecture

COMBlocker is designed to interpose itself in the instantiation of all COM objects. In terms of the attack described in Section 4.2, the COM object loaded by Microsoft Word would be matched against a security policy and when that COM object in turn attempts to load another COM object, that subsequent instantiation would also be checked against the policy.

To create such a policy enforcement system, the instantiation logic must be injected into every process on the system. This can be accomplished by a number of methods with varying levels of complexity and completeness. For reasons detailed later, COMBlocker injects a dynamically-linked library into every running process. This library contains the logic required to enforce the security policy. Once the COMBlocker library is injected into every process, calls to the COM instantiation APIs need to be redirected to the library to verify any object being loaded. We accomplish this through binary patching in our prototype. As the COMBlocker library is loaded, it locates the four COM instantiation APIs and overwrites the function prolog with a jump to the policy verification code.

With the binary hooks in place, any application that calls any of the COM instantiation APIs is redirected to the security interface. This redirection will take place regardless of the source of the call to the instantiation API. In other words, the call to load a COM object could come from a base executable, a library, a different COM object, or any other type of executable code. Any call to any of the instantiation APIs is verified against our security policy.

Once we can verify the instantiation of an object, a suitable policy must be defined. For the sake of simplicity our proof of concept starts by allowing the user to apply the Internet Explorer killbit list to all COM object instantiations. From there, a user can create exceptions to the killbit list or add disallowed objects on a per-application or system-wide basis.

5.3 Detailed Architecture

COMBlocker was developed as a third-party add-on for Microsoft Windows. It is not implemented by changing the underlying COM architecture or by invoking any extended APIs. If Microsoft were to implement a similar system, they could simply change the COM instantiation APIs to always check a central policy. The proof of concept developed in this research is a prototype of a system that a third-party could implement to provide a central COM security policy. With that, the details of how the solution was implemented are covered for completeness.

5.3.1 DLL Injection

To introduce the security policy verification code into every running process, COMBlocker uses DLL injection. This is the process by which an application forces another application to load a DLL of its choosing. There are several ways to inject a DLL into another process, but since our solution requires injection into every process, we chose to use the `AppInit_DLLs` registry key [18]. The operating system will load all DLL's specified in this key into every process, thereby providing comprehensive coverage for all applications.

5.3.2 Binary Hooking

Once the library is injected into every process, control flow from the COM instantiation APIs must be redirected to the COMBlocker security policy. Our proof of concept accomplishes this redirection through binary

Figure 5. Hooking Architecture for COMBlocker. Using binary rewriting, we force all COM objects to be checked against the global policy at their instantiation. Note on the left the policy check – `call AlertCLSID; test eax, eax;`

patching. The binary patching used by COMBlocker is very similar to the Detours API created by Microsoft Research [11]. In the case of COMBlocker, the function prolog of every COM instantiation API is overwritten with code that will jump to the security policy verification code in our library. As part of the DLLMain() function, our library will locate the four instantiation APIs and overwrite the first five bytes of the function prolog with a jump to the code that implements the policy verification. Since the first five bytes of the function have been overwritten, the first few instructions of the policy verification code must reproduce the functionality of the original API. Once these instructions are executed, the security policy enforcement can commence. Figure 5 shows the interception of control flow by COMBlocker to check against a global policy.

5.3.3 Enforcement Logic

Once the control flow has been redirected from the instantiation APIs to our own logic, we then have access to the arguments to those APIs. This provides us with the necessary information to create an enforcement policy. Specifically, the CLSID of the object being instantiated is passed to the instantiation APIs as the first argument. In our enforcement logic, we retrieve a pointer to the CLSID from the stack of the instantiation API. If the CLSID is specifically blocked by the security policy in the registry, the enforcement logic simply returns the error `REGDB_E_CLASSNOTREG` to the calling application. This returns an invalid handle that the calling application cannot use for interacting with the object. All applications tested gracefully handled this error condition, but if one did not, it would crash rather than allowing the exploit to continue.

5.3.4 Policy Definition

In COMBlocker, we wished to have a simple starting point to define the security policy. As such, the Internet Explorer killbit list can be applied to any application on the system or to the system as a whole. This is accomplished by setting a value in the registry in a location created by COMBlocker. The enforcement logic checks to see if this value is set. If it is, the killbit list is read from its default location in the registry and used for comparison against the CLSID the application is attempting to load. It is important to note that this is different than the security policy currently employed by Microsoft Word. In Word, the policy is only applied to the objects instantiated by the base executable. In our system, all instantiations are monitored.

Once the killbit list is applied to an application, exemptions or additions to the list can be manually entered. These modifications are also set in the COMBlocker registry hive. For each application, subkeys in the registry are used to define by CLSID which objects are specifically allowed or specifically denied. This allows the user

Figure 6. COMBlocker Successfully Stopping Instantiation

to create detailed white lists and black lists for every application or the system as a whole.

As mentioned in Section 3.3, it is unrealistic for a user to try to maintain lists of controls that should be considered secure and insecure. As such, it is the vision for a system like this to be implemented by Microsoft or a third party vendor. In these cases, Microsoft (or the vendor) could keep track of the known flawed controls and ensure they cannot be instantiated by any application except those specifically requiring them. Additionally, applications could be profiled to enumerate only those controls that should be instantiated under normal operation. In cases where this is possible, detailed white lists could also be created.

6 Results and Discussion

6.1 Breadth of Vulnerability

The first objective described in this paper was to determine the breadth of the vulnerability. The issue of transitive trust amongst COM objects was hypothesized to exist in all COM containers which load content provided by a third party. This proved to be true for every application that was tested. Throughout the research, the attack described in Section 4 was reproduced in Microsoft WordPad, Microsoft Excel, Microsoft Powerpoint and Adobe Reader. The question of how additional third party COM containers might behave was also approximated. Microsoft Visual Studio 6 ships with a utility called the ActiveX Control Test Container. This utility allows developers to test the functionality of COM objects without having to create their own COM container. This loosely represents how a generic COM container would behave under normal circumstances. The attack from Section 4 was also successful in this tool, meaning that a significant number of other third-party applications are also vulnerable to these attacks.

Each application tested was coerced into loading a COM object that was in direct violation of its security policy (where one exists). Generally speaking, these security policies are used to prevent the instantiation of COM objects that are known to contain vulnerabilities. In many cases, these security policies are used in lieu of fixing the vulnerabilities. With the research presented in this paper, each of the loitering vulnerabilities in those controls can be resurrected and used for successful compromise.

6.2 Effectiveness of the Solution

We tested COMBlocker's effectiveness and measured the overhead it imposes on a standard desktop system.

The first step in testing the effectiveness of the solution was to apply it to each of the applications that were found to be vulnerable in the section above. For each application that was shown to be vulnerable, COMBlocker presented the user with the dialog box shown in Figure 6. This dialog box shows that the application attempted to load a control that is specifically denied by the defined policy. It also indicates that the instantiation of the object was prevented. COMBlocker was successfully able to prevent the attack described in Section 4 in Microsoft WordPad, Word, Excel, and PowerPoint as well as the ActiveX Control Test Container.

Demonstrating the formal completeness of our solution is difficult. Our mechanism is helped by the fact that there are only a small number of publicly known

means by which COM objects can be instantiated. Injecting COMBlocker at these points should logically prevent applications from circumventing policy enforcement. However, if applications can instantiate COM objects through other unknown means such as implementing their own APIs, these interfaces would also need to be modified and mediated.

6.3 Performance

A version of COMBlocker was created that logged the time required for each policy lookup encountered throughout the operation of an application. The test build was installed on a typical development workstation and gathered information for all of the COM object instantiations that occurred during a single day as part of a developer's normal work. The test workstation was a Windows XP SP3 machine with Office 2007, Internet Explorer 7, Firefox 3, Lotus Notes 8, Visual Studio 6, and several other commonly installed applications.

During the course of the day, the behavior of the developer caused over 65,000 COM instantiations. Each of these recorded an average policy lookup time of 554μs to complete, with a 95% confidence interval of $\pm 104\mu$s. The variation in lookup time is largely due to the fact that consulting the killbit list in the registry is accomplished through a linear scan of the subkeys; it is not indexed. Testing shows that an average application incurs less than 10 policy lookups per user action. With that, each user action generates less than 5ms of delay due to COMBlocker.

Another data point gathered in this test is that in general, Office applications and web browsers incurred a lower lookup time than core operating system components. When the data set is reduced to only Office applications and web browsers, the average lookup time drops to 104μs, with a 95% confidence interval of $\pm 14.2\mu$s. This indicates that if performance were an issue in implementing a system like COMBlocker, the scope of the protection could be reduced to only those applications that are more easily targeted in COM-based attacks.

6.4 Discussion on Policy Creation

As mentioned in Section 5, the base policy for each application (or the system as a whole) was the Internet Explorer killbit list. It appears that over time, several COM objects required for the normal operation of many of the applications tested have been killbitted. In other words, there exist several COM objects that are critical to the operation of Microsoft Word, Excel, and even Internet Explorer that are in the killbit list. The question arises: How can COM objects critical to Internet Explorer end up in the killbit list? The answer is that the killbit list blocks the instantiation of COM objects by the IE scripting engines, not the base IE executable. An example of this occurred when COMBlocker applied the killbit list to Internet Explorer as a whole, the navigation bar was prevented from being instantiated.

The takeaway from this result of the testing is that the killbit list cannot be blindly applied to every application or the system as a whole. As policies are created, the killbit list can be used as a base, but modifications are required for each application being monitored. Any entity choosing to provide a solution like COMBlocker must take great care to ensure the security policies they define will allow for the normal operation of each monitored application while still providing a suitable level of security.

6.5 Future Enhancements to COMBlocker

As mentioned above, policy creation can be quite complex. While the killbit list is a good starting point for a blacklist, it simply does not apply as-is to every application on the system. A useful enhancement to COMBlocker (or an accompanying tool) would be one that allows for the profiling of applications under normal use. This would help to expedite the identification of controls listed in the killbit list that are critical to the operation of other applications.

Additionally, more research is required to determine the feasibility of runtime analysis of persistence streams. It could be possible to analyze data contained in a persistence stream and filter access to that data based on policy. For example, one policy could be to ensure that COM objects may only load simple data types and cannot load the more complex types, which could force the instantiation of other objects.

6.6 Comparsion to Microsoft-Issued Patch

On June 8, 2010, Microsoft released Security Bulletin MS10-036 in response to our private disclosure of this vulnerability. This patch attempts to prevent the attacks discovered and demonstrated in this paper. Specifically, this patch extends the killbit list to nested COM instantiations made by Microsoft Office. While this patch effectively prevents Office-generated files from including these attacks, it does not protect any other application that takes advantage of the COM infrastructure. Accordingly, this patch does not provide the security guarantees of COMBlocker, and therefore means that Windows-based systems still remain vulnerable to such attacks through other applications.

7 Conclusion

COM has provided developers with an extremely flexible and extensible framework for the creation and use of myriad software components. However, we have shown in this paper that a lack of central security policy for the instantiation of COM objects allows a single flawed control to pose a security threat to all applications relying on the COM infrastructure and the system as a whole. Moreover, because the average Windows installation can have hundreds of known flawed controls registered, COM provides attackers with an extremely wide range of vulnerable surfaces through which to compromise a system. We have demonstrated that each of the per-application security policies for several popular applications can be bypassed – each allowing for the instantiation of any of the hundreds of known flawed controls. In response, we have developed and characterized the reference monitor-like COMBlocker, which interposes itself on all COM instantiations to ensure that such operations conform to a global policy. In so doing, we have significantly improved the resistance of the COM architecture and applications relying upon it to transitive trust-based exploits.

Acknowledgments

We would like to thank Ryan Smith for his contributions to the research presented in this paper. Additionally, we thank Reiner Sailer and William Enck for their help in providing early reviews of this work. This publication represents the views of the authors and does not necessarily represent IBM's positions, strategies, or opinions.

References

[1] V. Anupam and A. Mayer. Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies. In *Proceedings of the USENIX Security Symposium (SECURITY)*, 1998.

[2] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *ACM Conference on Programming Language Design and Implementation*, June 2005.

[3] H. Chen and D. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2002.

[4] M. Dowd, R. Smith, and D. Dewey. Attacking Interoperability. Proceedings of Black Hat 2009, July 2009. `https://media.blackhat.com/bh-usa-09/video/DOWD/BHUSA09-Dowd-AtkInterop-VIDEO.mov`.

[5] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, January 2004.

[6] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*, May 1999.

[7] T. Fraser, N. L. Petroni, and W. A. Arbaugh. Applying Flow-Sensitive CQUAL to verify MINIX Authorization Check Placement. In *Proceedings of the Workshop on Programming Languages and Analysis for Security*, 2006.

[8] V. Ganapathy, T. Jaeger, and S. Jha. Automatic Placement of Authorization Hooks in the Linux Security Modules Framework. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2005.

[9] V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy (OAKLAND)*, 2006.

[10] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining Security-Sensitive Operations in Legacy Code Using Concept Analysis. In *Proceedings of the Internation Conference on Software Engineering (ICSE)*, 2007.

[11] G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. In *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[12] N. Islam, R. Anand, T. Jaeger, and J. R. Rao. A flexible security system for using internet content. *IEEE Software*, 14:52–59, 1997.

[13] T. Jaeger, A. Edwards, and X. Zhang. Consistency Analysis of Authorization Hook Placement in the Linux Security Modules Framework. *ACM Transactions on Information and System Security (TISSEC)*, 7(2), 2004.

[14] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable context. In *Proceedings of the USENIX Security Symposium (SECURITY)*, 1996.

[15] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, and R. C. Taylor. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the National Information Systems Security Conference*, 1998.

[16] D. Malkhi and M. Reiter. Secure Execution of Java Applets Using a Remote Playground. *IEEE Transactions on Software Engineering*, 27(12):1197–1209, 2000.

[17] Microsoft. Safe Initialization and Scripting for ActiveX Controls. Microsoft Developer Network. `http://msdn.microsoft.com/en-us/library/aa751977(VS.85).aspx`.

[18] Microsoft. Working with the AppInitDLLs registry value. Microsoft Support, November 2006. `http://support.microsoft.com/kb/197571`.

[19] Microsoft. Microsoft Security Advisory (953839). Microsoft TechNet, August 2008. `http://www.microsoft.com/technet/security/advisory/953839.mspx`.

[20] Microsoft. Microsoft Security Advisory (956391). Microsoft TechNet, October 2008. `http://www.microsoft.com/technet/security/advisory/956391.mspx`.

[21] Microsoft. How to stop an ActiveX control from running in Internet Explorer. Microsoft Support Center, August 2009. http://support.microsoft.com/kb/240797.

[22] Microsoft. Microsoft Security Advisory (960715). Microsoft TechNet, February 2009. http://www.microsoft.com/technet/security/advisory/960715.mspx.

[23] Microsoft. Microsoft Security Advisory (969898). Microsoft TechNet, June 2009. http://www.microsoft.com/technet/security/advisory/969898.mspx.

[24] Microsoft. Microsoft Security Bulletin MS09-032. Microsoft TechNet, July 2009. http://www.microsoft.com/technet/security/bulletin/ms09-032.mspx.

[25] Microsoft. Microsoft Security Bulletin (MS09-034). http://www.microsoft.com/technet/security/bulletin/ms09-034.mspx, July 2009.

[26] Microsoft Support. You are prompted to grant permission for ActiveX Controls when you open an Office XP or Office 2003 document. Microsoft Support Center, October 2007. http://support.microsoft.com/default.aspx?scid=kb;en-us;827742.

[27] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[28] N. Thompson. MFC/COM Objects 1: Creating a Simple Object. Microsoft Developer Network, March 1995. http://msdn.microsoft.com/en-us/library/ms809986.aspx.

[29] J. Yang, T. Kremenek, Y. Xie, and D. Engler. MECA: An Extensible, Expressive System and Language for Statically Checking Security Properties. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2003.

[30] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the USENIX Security Symposium (SECURITY)*, 2002.

8 Appendix A

The following source code represents the CComVariant::ReadFromStream() method as distributed with ATL version 6.0. This is the method that allows a developer to read a variant from a COM persistence stream. The key piece is the call to OleLoadFromStream() (line 23) which is detailed in Appendix B:

```
1 inline HRESULT CComVariant::ReadFromStream(IStream* pStream)
2{
3    ATLASSERT(pStream != NULL);
4    HRESULT hr;
5    hr = VariantClear(this);
6    if (FAILED(hr))
7        return hr;
8    VARTYPE vtRead;
9    hr = pStream->Read(&vtRead, sizeof(VARTYPE), NULL);
10    if (hr == S_FALSE)
11        hr = E_FAIL;
12    if (FAILED(hr))
13        return hr;
14
15    vt = vtRead;
16    int cbRead = 0;
17    switch (vtRead)
18    {
19    case VT_UNKNOWN:
20    case VT_DISPATCH:
21        {
22            punkVal = NULL;
23            hr = OleLoadFromStream(pStream,
24            (vtRead == VT_UNKNOWN) ? IID_IUnknown : IID_IDispatch,
25            (void**)&punkVal);
26            if (hr == REGDB_E_CLASSNOTREG)
27                hr = S_OK;
28            return S_OK;
29        }
```

9 Appendix B

The following code was derived by first disassembling Ole32.dll with IDA Pro. From there, the disassembly was manually converted to the C/C++ equivalent shown below. This code shows that this method will read a CLSID from a persistence stream and instantiate that object with a call to CoCreateInstance() (line 23). It should be noted that at no time is there a check against a security policy before the object is loaded:

```
1 HRESULT OleLoadFromStream(LPSTREAM pStm,const IID *const iidInterface,
  LPVOID *ppvObj)
2{
3    CLSID pclsid;
4    IID *riid;
5    HANDLE *ppvObj;
6    HANDLE *ppvStmObj;
7    HRESULT hr;
8
9    riid = iidInterface;
10
11   if (*ppvObj == 0)
12        return E_INVALIDARG;
13
14    . . .
15
16    if (!isValidInterface(pStm))
17        return E_INVALIDARG;
18
19    hr = ReadClassStm(pStm, &pclsid);
20    if (hr)
21        return hr;
22
23    hr = CoCreateInstance(pclsid, NULL, CLSCTX_NO_CODE_DOWNLOAD|
24            CLSCTX_REMOVE_SERVER|CLSCTX_LOCAL_SERVER|CLSCTX_INPROC_SERVER,
25            riid, ppvObj);
26    if (hr)
27        return hr;
28
29    hr = ppvObj->QueryInterface(IID_IPersistStream, ppvStmObj);
30    if (hr) {
31        ppvObj->Release();
32        return hr;
33    }
34
35    hr = ppvStmObj->Load(pStm);
36
37    ppvObj->Release();
38
39    if (hr) {
40        ppvObj->Release();
41        return hr;
42    }
43
44    . . .
45
```

```
46    return hr;
47}
```